https://doi.org/10.48009/1_iis_2013_61-71

Issues in Information Systems

Volume 14, Issue 1, pp.61-71, 2013

HARDWARE SECURITY TAGS FOR ENHANCED OPERATING SYSTEM SECURITY

Jia Song and Jim Alves-Foss
Center for Secure and Dependable Systems
University of Idaho, Moscow, Idaho USA
song3202@vandals.uidaho.edu, jimaf@uidaho.edu

ABSTRACT

Most modern software is vulnerable to attack from a wide range of sources. To assist the system developer, researchers are looking at hardware-based security tagging schemes to enhance system security. This paper addresses the design and implementation of a new tagging scheme for access control and information flow; specifically the implementation at the assembly language level for a zero-kernel operating system. We also discuss key lessons learned that we have not seen addressed in related literature.

Keywords: Security Tagging, RTEMS, SPARC, DIFT, Zero-Kernel Operating System (ZKOS)

INTRODUCTION

Researchers and hardware developers have been exploring techniques to enhance the security of our computer systems. They are painfully aware that even the best software developers make mistakes, and with an internet user base of billions of interconnected people, there are just too many opportunities for our systems to come under attack. One of the technologies being explored involves the use of a data tagging scheme. These schemes attach security labels to memory regions and processor registers to carry information about tagged data during program execution. They can be used to ensure that the semantics of computations are correctly implemented; to isolate code and data, users and system; and they can be used to enforce security policies at the hardware level. These hardware enhancements relieve the software developer of some of the more burdensome security programming concerns and provide improved performance, as compared to traditional microprocessors.

We have developed one of these tagging schemes and implemented it on the Open SPARC microprocessor [12]. Our initial implementation was based on 'C' programming language features [13], and then was mapped down to the SPARC assembly language level and modeled in a simulator. During this mapping and implementation we learned a few things that have not been discussed in the prior literature, justifying our look at the assembly language version of tagging instead of just the higher level C-language tagging rules.

This paper provides a brief discussion of related research and the overall concepts of this project, including the ideas of a zero-kernel operating system. It describes the design of a new tagging scheme, explains the tagging rules for SPARC instructions, and provides an assembly code example to demonstrate how the tags are checked and propagated. We conclude with lessons learned when mapping the C language constructs down to assembly language; including problems associated with compiler optimizations which made testing difficult, and special hardware features such as the SPARC register window, which changes the standard parameter passing conventions.

BACKGROUND

We evaluated published security tagging approaches, and divided them into two categories: use of security tags for attack prevention and use of security tags for access control (see [12] for details).

Tagging Approaches for Attack Prevention and Access Control

Attack prevention tagging techniques are based on the idea of Dynamic information Flow Tracking (DIFT) [4, 6, 9, 14]. System software programs the hardware to tag user-provided data, and then the hardware propagates those tags and checks to ensure that the data is not being used for an attack against the program, (e,g. causing a buffer

overflow). Improper use of tagged data generates a security exception, and the system is responsible for managing the exception. Other approaches use improvements to compilers or analysis tools to implement the tagging [7, 16].
In addition to tagging for information flow, tagging has been used for access control, providing a fine-grain protection of system data and code [10, 15, 17]. Hardware is programmed by the system software, indicating permissions of data/code; and checks if the executing program is violating those permissions. Software is involved in configuring the system, handling the exception and sometimes handling the checks and propagation rules. Trust-management, intrusion-tolerance, accountability, and reconstitution architecture (TIARA) [11] is a co-design of hardware, operating system, and applications. TIARA implements access control at different levels. At the hardware level, the tag management unit enforces fine-grained access control for individual words in memory and registers. At a higher level, another access control mechanism is imposed. TIARA is an example of a zero-kernel operating system (ZKOS) in that it has separate OS modules that run in user mode, protected by hardware tagging.

Zero Kernel Operating System

Traditional memory protection, protected mode, and supervisor mode gives the programs running in supervisor mode ultimate privileges and the programs running in protected mode limited privileges. Different from the protected mode and supervisor mode, Shrobe, DeHon, and Knight [11] indicate that a ZKOS decomposes the whole operating system into many smaller components, each having its own limited privileges. This ensures that the component has the least privileges needed. A ZKOS also prevents any of the components from having complete authority. Therefore, even if a component is compromised, the whole system will not be compromised. By using smaller components, the system provides better isolation for code and data and also a better separation of system and users. Supported by a security tagged architecture, a ZKOS would be more flexible and powerful. This approach avoids the expensive context switches between kernel mode and user mode that occur in traditional operating systems because the components have their own hardware recognized privileges and compartments. What is more, the decomposed components help ensure fine-grained memory protection of the system.

RTEMS

RTEMS [1] is a real-time executive that provides a powerful runtime environment, allowing various types of services and applications to be embedded. It supports a complex set of services that include multitasking, inter-task communication, and dynamic memory allocation. RTEMS is designed with a single-user multiple thread execution model, and therefore does not have concepts of different users or support for separation of users from the core "operating system". This made it a prime target for initial evaluation of a hardware-based security tagging scheme that provides fine-grain access controls, whose structure follows the concept of a ZKOS [11].

RTEMS has 18 managers that provide services (through the use of directives) to user code in support of concepts such as tasks, memory, timers, and communications. Each manager also has internal functions that support implementation; some of these are private to the specific manager, and some are intended to be used by other managers. In either case, these internal functions are not intended to be used by user code (Fig. 1). In addition, RTEMS has a set of modules that make up the SCORE subsystem (Super Core). The SCORE provides services for all managers, and all managers interact with the SCORE via directives. SCORE modules are essential to the internal working of RTEMS, but are not intended for use by user code. The tagging scheme must enforce these restrictions.

NEW SECURITY TAGGING SCHEME DESIGN

Our tagging scheme uses a 3-part tag consisting of an owner field, code-space field, and control-bits field. Each of these fields helps maintain correctness and security for RTEMS. Although our new tagging scheme was designed to provide security for RTEMS, the issues addressed are common to many types of operating systems and thus can be usable for them as well. Our tagging scheme allows us to use the principle of least privilege in protecting resources in the system by providing a finer-granularity than traditional supervisor and user modes, protection rings or memory management units. We can provide many security domains and separately protect individual words of memory. We assume the existence of a hardware tag engine that enforces the tag checking and propagation rules generates an exception when the checks fail.

Tag Format

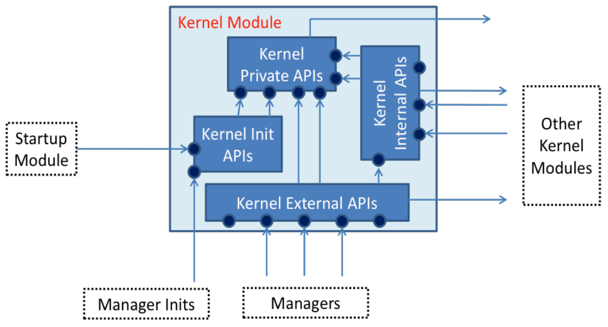


Figure 1. RTEMs Modules and Connectivity

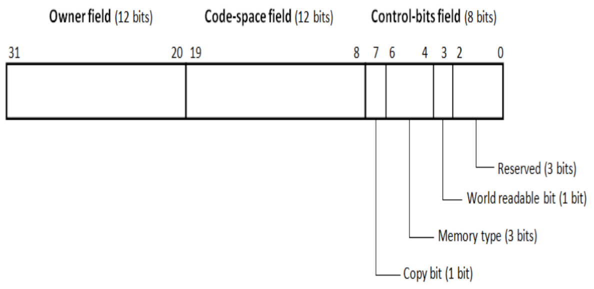


Figure 2. Tag format

Fig. 2 depicts the format of our tags, consisting of three fields: Owner, Code-space, and Control-bits. The tag can be written as (<Owner>, <Code-space>, <Control-bits>). The Owner field separates the code/data of individual system modules and separate users. The intent of this field is to indicate the intended ownership of data. For example, when the thread manager creates a thread for a user, it also creates internal data structures related to that thread. The Owner field indicates the user that "owns" the thread and helps prevent one user from asking the thread manager to modify a thread owned by another user.

The values of the Owner field are classified into six major classes based on the structure of RTEMS: SCORE internal, SCORE, Manager internal, Manager, Startup and User. We further divided the SCORE internal class into SCORE internal init, SCORE private internal, and SCORE internal groups. The SCORE internal functions are shared among different SCORE modules. The SCORE private internal and SCORE internal init functions are only used by the corresponding SCORE module. The Manager internal class is also divided into similar groups.

By using the owner field, the owner of data or code can be easily identified. For example the tag (**task manager**, <Codes-pace>, <Control-bits>) shows that the data or code is owned by the task manager. In the remaining sections, <SCORE> in the tag means this field could be one of the possible values in the SCORE class, and the same holds for other classes. Although RTEMS currently only supports a single user, our plan is to expand it to a multiuser system. Therefore, we list multiple users in the possible values of the User class. By having different users for the Owner field, we can ensure that a user can only access his own data and code, but not other users' resources.

The Code-space field is used to show the class of the code or data and helps control function calls. For data, this field indicates which operating system modules are allowed to access the data. For example, the tag (User1, Manager1, <Control-bits>) means the data is created in the manager1's code for user1. With executable code, we use Code-space to indicate which class the code belongs to, and then provide rules to control which classes of code can use which other classes of code; thus preventing users from calling privileged internal operating system functions. This make the Code-space field critical for information flow control and memory access control. The Code-space field can be <User>, <Manager>, <Manager internal>, <SCORE>, <SCORE internal> and <Startup>.

The 8-bit Control-bits field is used for further control. We use a copy bit (bit 7) to indicate whether a return value has been modified. The copy bit allows user code to have a copy of a trusted data value (i.e., a task ID) as long as it is not changed. The notation $\overline{cp}$ indicates that the copy bit is not set while *cp* indicates the copy bit is set. The value returned to a user will be tagged with the security class of the directive and will have the copy bit set. If the copy bit remains set, it means that user has not made any change to the value. If the copy bit is not set, the data is treated as modified data and will not be accepted when used as a parameter to a directive. For example, if user1 uses directive code from task manager to get a task ID, the directive returns an ID which is tagged (user1, task manager, *cp*). If the user modifies this returned ID, the tag will be changed to (user1, user1, $\overline{cp}$) to indicate that the ID has been changed; and the OS will no longer trust the ID. This allows the system to trust IDs that come from users without having to keep an internal table of indirect identifiers, and therefore improving performance and simplifying system code.

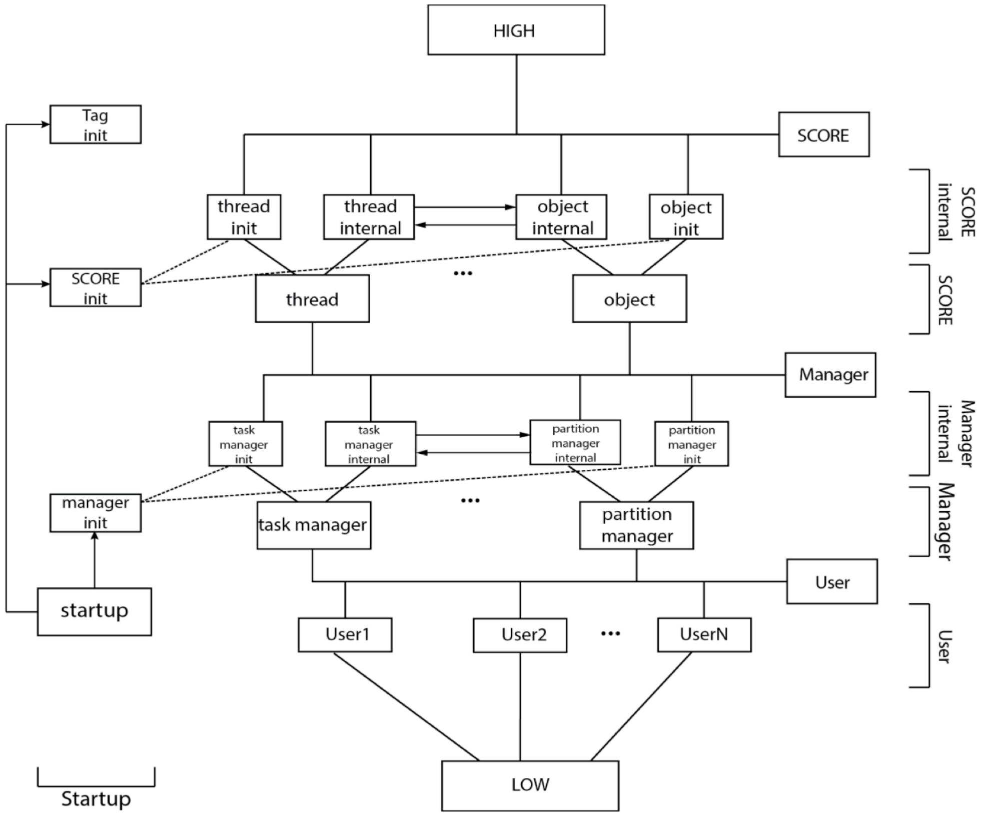
We allocate three bits (bit 6 to 4) for memory type, divided into three classes: stack memory, code memory, and data memory. All memory can be marked as read-only. Normally only stack memory and data memory can be writable. Among the 3 bits, we use bit 6 to indicate it is read-only or readable and writable memory (1 indicates readable and writable and 0 indicates read-only). Code memory stores executable and readable code. The "entry point" of a function has a special tag to indicate the correct starting address for executing a function. Therefore for the lower 2 bits, 11 indicates an entry point to an executable function, 10 means the memory is executable but not an entry point, 01 indicates stack memory while 00 indicates data memory. We use a world-readable bit (bit 3) to indicate that the tagged data can be read by all entities; used when the system (higher level) wants to give the user (lower level) permission to access some data, such as configuration data. We reserve the remaining 3 bits (bit 2 to 0) for future use (e.g., DIFT-style protection [14]).

The Lattice and Our Tagging Scheme

Since the Owner field and Code-space field are used to define the security class of the data, we can implement ideas similar to those of the Data Mark Machine (DMM) [5] to control the information flows within RTEMS. Our solution is a bit different from DMM since we are defining a hierarchy with confidentiality and integrity controls, and not for traditional multi-level security. However, we still can control access preventing "lower-level" entities from unauthorized access to higher level entities.

We defined two parallel lattices for the information flow control within RTEMS, for each of the Owner and Code-space fields of the tags (Fig. 3). To simplify the diagram, we assume that boxes higher on the figure have higher integrity and security classes than those below them, and that the ≤, ≥ and ⊕ operators support that hierarchy. This dual-use (security and integrity) is different from traditional security classes, but maps well to the standard use of privilege rings or supervisor/user execution modes in hardware.

In our model the security class of an entity, a, will be denoted as a̲, and it can be written as a̲ = (Owner(a), Code-



space(a)). We have ignored the Control-bits in this discussion since they are used separately from the lattice-based controls and formulas. The definition of the least upper bound, ⊕, of the security classes of two tags is: if $\underline{a}$ = (Owner(a), Code-space(a)), and $\underline{b}$ = (Owner(b), Code-space(b)), then $\underline{a} \oplus \underline{b}$ = (Owner(a) ⊕ Owner(b), Code-space(a) ⊕ Code-space(b)). We have similar rules for ≤, ≥, >, < and = operations.

Figure 3. Our security lattice

SPARC Architecture

This project implements the tagging scheme on a SPARC processor; therefore we provide a brief discussion of SPARC here. SPARC has an instruction set architecture that is derived from a reduced instruction set computer (RISC) lineage. The difference between SPARC and RISC is that SPARC provides flexible register window management by using separate instructions for window management and function call and return.

At any given time, an instruction can see 8 global registers (%g0-%g7) and a register window which contains 24 registers: 8 in registers (%i0-%i7), 8 local registers (%l0-%l7) and 8 out registers (%o0-%o7). In and out registers are used for passing parameters and getting return values, because the caller's out registers become the callee's in register when there is a normal function call. The CALL instruction does not cause this register mapping, rather the SAVE instruction slides the register window -- allowing more flexibility for the compiler, but decoupling parameter passing from the actual function call. The RESTORE instruction is used to slide the window back before a return. The first six in registers (%i0-%i5) are used to store incoming parameters. %i6 is for storing the frame pointer (%fp) and %i7 is for storing the return address. Among the out registers, %o0-%o5 are used to store the parameters being passed to a called subroutine, %o6 stores the stack pointer, and %o7 stores a temporary value or the return address. Registers %l0-%l7 are local registers that are mostly used for temporary values. The current PC and the next PC (nPC) are copied to %l1 and %l2 when a trap occurs. The global registers have global scope. The %g0 register has the hardwired value of zero, therefore any write to it has no effect.

In the SPARC architecture, partially overlapping windowed integer registers are provided. A window contains 8 each of input, output and local registers. The current window pointer (CWP) indicates the current window and walks a circular buffer of windows. The execution of SAVE or trap instructions decreases the CWP by 1 and RESTORE or RETT increases the CWP by 1. The overlap of windows allows each window to share its in and out registers with adjacent windows. When the CWP decreases by one, the out registers of the previous register window become the in registers of the new register window and the caller's stack pointer (%sp) becomes the current procedure's frame pointer (%fp). When executing RESTORE or RETT, the CWP increases by one and the window moves back to the previous window. The local registers are unique to each window. A return value is stored in %i0, therefore, after moving the register window with a RESTORE, the caller can access the return value that is stored in its out register %o0, which was callee's in register %i0. Window overflows and underflows cause a trap to the operating system.

TAGGING RULES FOR SPARC INSTRUCTIONS

This section explains the implementation of our tagging scheme at the assembly language level. During our study of SPARC instructions, we divided instructions into five major groups and have tagging rules for each of these groups.

Rules for branch instructions

Branch instructions are either unconditional branches or conditional branches. Unconditional branch instructions are BA (branch always), which means always execute the branch, and BN (branch never), which means never execute the branch (effectively a NOP). Conditional branches are based on the condition codes, which are updated by compare instructions, to decide whether or not take the branch. Therefore, we need to check whether or not the current running thread is permitted to access the condition code. For the execution of the target code, a check is needed to make sure the current thread is allowed to execute (jump to) the target code. Consequently, for branch instructions such as Bicc address, the tags of the address and PC must be checked to ensure the jump is appropriate. Since this is a jump, we want to make sure we don't change Code-spaces. Therefore, the Code-space(address) (Code-space(address) refers to the Code-space field of the addresses tag) and Code-space(PC) must be checked to make sure the current thread is allowed to execute the target code. In addition, we need to check and ensure that the target address is executable code memory, but not data memory or stack. The rule for branch instructions is: if [condition code] ≤ PC, and Code-space(address) = Code-space(PC), and the target address is an executable code memory then the branch is allowed; otherwise, throw an exception.

Rules for call related instructions

The four call related instructions are CALL, JMPL, RETT and Ticc. When making a function call (CALL address), the tag system must check whether or not the function is allowed to be called by the current program using our function execution control rules. Our function execution control rules specify that if the current program is not allowed to call the function, then the call will not be executed. If it is allowed under our rules and the target code is the entry point of an executable function, then the call is allowed. The call will save the value of PC into %o7, PC into %o7, set the copy-bit of %o7, update the PC and tag of the PC, and then start execution of the function. The tag of the updated PC will follow the function execution control rules. The rule for CALL instruction is: if the function call is allowed under the function execution control rule, and if the target code is tagged as entry point to an executable function, then the CALL instruction is allowed.

The JMPL instruction is used to save the current PC in a specified register and then jump to any specified address. There are two usages of JMPL. The first is JMPL address, %g0 which copies the current PC to %g0 before making the jump. However, because %g0 always has value 0 in it and cannot be overwritten, the write to it has no effect. As a result this is a true jump with no expected return. As with branch instruction, we need to make sure that the current running program has the permission to jump and execute the target code. For that reason the security class of PC and the target address must be compared. The rule for the first usage of the JMPL instruction is: if Code-space(address) = Code-space(PC), and the target address is an executable code memory then the JMPL is allowed.

The second usage, JMPL address, %o7 will write the current PC to %o7. Since SPARC uses JMPL %o7+8, %g0 to jump back according to the return address stored in %o7, a check is needed to ensure the return address has not been modified. However, JMPL has many other usages, so we cannot give JMPL such a restrictive rule. Consequently we require the implementation of a new instruction, RET, which is used for return from subroutines. The format of RET instruction could be RET %o7+8, %g0 (SPARC specifies a RET instruction that is just a shorthand notation for JMPL %o7+8, %g0, we will replace this with a separate instruction). For return from a subroutine, if the copy-bit of the tag of %o7 is not set, then the return is not allowed. If the copy-bit is set, which ensures that the return address was not modified, then we need to check the PC and the return address stored in %o7 and also the PC and %o7. The check is to ensure that the return address has not been modified. The rule for RET is: when the copy-bit of %o7 is not set, the RET instruction is not allowed. If the copy-bit of %o7 is set, and Code-space([%o7+8]) = Code-space(%o7) and the target address ([%o7+8]) is an executable code, then the RET is allowed, with the new PC = %o7+8, and the security class of the tag of PC is %o7.

RETT is used to return from a trap handler. Ticc is used to generate a trap to the trap handler based on an integer condition code. For this initial tagging system, we apply the same rules we apply to CALL and RET for these instructions; understanding that they will have to be modeled and possibly changed for a fully secure system.

Rules for arithmetic, logic and shifting instructions

For arithmetic instructions, logic instructions and shifting instructions, we propagate tags. For example, the instruction ADD %g2, %g1, %g3 adds the value in %g2 to the value in %g1, then stores the result in %g3. To propagate tags, the copy-bit in the tags of the value in %g1 and %g2 need to be checked, and the result value stored in %g3 has a new tag associated with it. Whether or not the result can be stored in %g3 is not checked, instead we use registers as temporary storage. Taking ADD %g2, %g1, %g3 as an example, the new tag of %g3 is calculated based on the tags of %g2 and %g1, with the copy-bit unset. The particular rules for arithmetic, logic, and shifting instructions are shown in Table 1. Instructions that modify the condition code, such as ANDcc, ORcc, SUBcc, and UMULcc, set the tag of the condition code to the tag of the result. For example, after executing the instruction ADDcc %g2, %g1, %g3, the condition code's tag will be the tag of %g3. If %g1 is an immediate value, then %g1's tag is PC's tag with the copy-bit not set.

Table 1. Rules for arithmetic, logic and shifting instructions

Copy-bit of %g2	Copy-bit of %g1	Security class of %g3	Copy-bit of %g3
cp	cp	PC	$\overline{cp}$
cp	$\overline{cp}$	%g1	$\overline{cp}$
$\overline{cp}$	cp	%g2	$\overline{cp}$
$\overline{cp}$	$\overline{cp}$	%g1 ⊕ %g2	$\overline{cp}$

Rules for load and store instructions

Load instructions are used to load a value from a memory space and store it to a register. For example, the instruction LD [%fp], %o1 loads the content of the memory address [%fp] to the %o1 register. A check of whether the current program can read and use the value is needed. What is more, we need to check that the current running thread can access the value of %fp. Therefore the security classes of PC and the tag of the value and the tag of the %fp need to be checked. The load instruction is allowed when $\underline{[\%fp]} \leq \underline{PC}$, where $\underline{[\%fp]}$ denotes the security class of the tag of the value stored in [%fp]. The rules for load instructions enforce these checks of expressions and parameters. The rules for load instructions (LD [%fp], %o1) are:

(i) If the copy-bit in the tag of [%fp] is cp, and $\underline{\%fp} \leq \underline{PC}$, and Owner([%fp]) ≤ Owner(PC), and [%fp] is readable memory, then the load is allowed and the tag of the data will be copied to the register's tag.

(ii) If the copy-bit in the tag of [%fp] is $\overline{cp}$, and if $\underline{[\%fp]} \leq \underline{PC}$ and $\underline{\%fp} \leq \underline{PC}$, and [%fp] is readable memory, then the load instruction is allowed and the tag of the data will be copied to the register's tag.

Store instructions are used to store the value from a register to memory. The rules for store instructions check whether or not the value of the register is allowed to be stored in that location. For example, for the instruction ST %g1, [%fp], which stores the content of %g1 to memory space [%fp], we need to make sure the current running thread can access the value of %fp and write to the memory space. Rules for store instructions (ST %g1, [%fp]) are:

(i) When the copy-bit in the tag of [%fp] is $\overline{cp}$ and the copy-bit in the tag of %g1 is $\overline{cp}$, if $\underline{[\%fp]} \leq \underline{PC}$ and $\underline{\%fp} \leq \underline{PC}$ and [%fp] is writable data memory, the store instruction is allowed. The tag of the [%fp] is unchanged.

(ii) When the copy-bit in the tag of [%fp] is $\overline{cp}$ and the copy-bit in the tag of %g1 is cp, if $\underline{[\%fp]} \leq \underline{PC}$ and $\underline{\%fp} \leq \underline{PC}$ and Owner(%g1) = Owner([%fp]) and [%fp] is writable data memory, the store instruction is allowed. The tag of the [%fp] is copied from the tag of %g1.

(iii) When the copy-bit in the tag of [%fp] is cp and the copy-bit in the tag of %g1 is $\overline{cp}$, if Owner([%fp]) ≤ Owner(PC) and $\underline{\%fp} \leq \underline{PC}$ and [%fp] is writable data memory, the store instruction is allowed. The Code-space of the tag of the [%fp] will be reset to the Owner field of the tag and the copy-bit will be unset.

(iv) When the copy-bit in the tag of [%fp] is cp and the copy-bit in the tag of %g1 is cp, if Owner([%fp]) ≤ Owner(PC) and $\underline{\%fp} \leq \underline{PC}$ and Owner(%g1) = Owner([%fp]) and [%fp] is writable data memory, the store instruction is allowed. The tag of the [%fp] is changed to the tag of %g1. If [%fp] is writable stack memory, then write is allowed and tag of [%fp] is tag of %g1.

Rules for SAVE and RESTORE instructions

The SAVE and RESTORE instructions are used to slide the register window between the caller and callee windows. It is possible but rare, to use these instructions without a corresponding subroutine call. The execution of the SAVE instruction causes the system to subtract one from the CWP. Therefore, the instruction automatically allocates a new window of registers and a new stack frame in the main memory. The out registers of the caller become the in registers of the callee. The SAVE instruction acts like an ADD instruction, which adds a number to %sp. The number indicates the size of the stack for the new function. The caller's stack pointer %sp automatically becomes the frame pointer %fp of the callee. The instruction RESTORE is used to restore the caller's window. The instruction adds one to CWP, therefore the callee's in registers become the caller's out registers.

Since the user can use SAVE and RESTORE instructions, we are concerned about the security of the windows. The user may use the RESTORE instruction to change the register window to get data used in the caller's subroutine. Therefore, we add one tag to every register window to indicate who created the register window. On every SAVE instruction, we copy the tag of the PC to the tag of the new window. On every RESTORE instruction, we check the tag of the caller's window. If the tag indicates that creator of the window is not the one who wants to restore (pop) the window, then the restore instruction is not allowed to execute. During traps and exceptions, we want to give the SCORE code the ability to manage the register window. As a result the SCORE code has the permission to restore any window or register without any check of the window's tag. If the tag of the register window is identical to the tag of the PC, then the restore is allowed.

The rules for SAVE and RESTORE instructions are: On every SAVE instruction, the tag of the PC will be copied to the tag of the register window. For the RESTORE instruction, if the tag of the register window and the PC's tag are same or if Code-space(PC) > MANAGER, then the RESTORE is allowed. Otherwise the RESTORE is not allowed.

Concerns about the memory and stack in SPARC architecture

Assembly language implementation of the C programming language requires that we allocate memory to store local variables, functions, parameters, and linkage information in support of procedure calls. This memory region for each procedure is called a frame. Normally, these frames are allocated on a runtime stack, and pushed and popped with procedure calls and returns. For the SPARC architecture, the frame contains space for registers, parameters, PC, and so on. To access the frame, %fp, the frame pointer is normally used.

Stack memory is used to store frames, and the frames on the same stack in a ZKOS can belong to directive code or users. Therefore, we cannot treat frames the same as regular memory. For example, after return from a directive, the memory that has been used for that directive frame will be tagged with a tag containing Manager code-space. If the user tries to call a new user function, we could not use the same memory for the user function frame without violating our assignment rules unless we treat the stack differently from regular memory. Stack memory is treated as if the copy-bit is set for all write access to the stack, and must therefore be limited to specific pages of memory.

Sample assembly code

Fig. 4 shows simple C-code (lines 2-15), compiled with sparc-rtems-gcc, which is a cross compiler of C-code for RTEMS running on SPARC architecture, to generate assembly code (lines 17-45). Assume the main function is user code, and the foo function is directive code that has a tag (manager1, manager1) associated with it.

We starting execution at line 18 in the assembly program with the PC tagged (user1, user1). Line 18 sets the stack frame for the main() function by allocating save space for all registers and space for the local variables a, b, c. Following the stack-frame conventions, the local variables will be stored "below" the frame pointer with a at location [%fp-20], b at [%fp-16] and c at [%fp-12]. Note that this convention assumes that a is at a lower location in memory than the other local variables, effectively as if it was pushed on the stack last. At this point we are treating the memory as a stack. Line 19 assigns a constant to global register %g1, giving %g1 the tag of the PC (completing the right-hand side of line 4, evaluating the expression and giving the expression a tag). Line 20 then stores the value into memory location for a after completing the assignment tagging rules (completing the a=25 from line 4). Lines 21 and 22 perform similarly for b=7. Lines 23 and 24 evaluate the parameters of the function call from line 7. The load checks ensure that the current thread is permitted to read the values and puts them in registers, in this case inheriting the tag of the parameters. Line 25 performs the function call using the function execution rules (completing the call of line 7). The new tag of PC is (user1, manager1).

Line 34 allocates a stack frame on the stack for foo() and allocates a new register window with the tag of (user1, manager1). Line 35 stores the parameter a, which is from %i0 to the space [%fp+68] in the stack. Line 36 stores the parameter b to the space [%fp+72] in the stack. Note that both of the parameters passed to the foo() function are from the in registers, which were the out register of the main function. Rules for store instructions are checked for the ST instruction on line 34 and 35 (completing the parameter pass).

```
1   C code:
2   main()
3   {
4     int a=25;
5     int b=7;
6     int c;
7     c=foo(a,b);
8   }
9
10  int foo(int a1, int b1)
11  {
12    int c1;
13    c1=a1+b1;
14    return c1;
15  }
```

```
16      Assembly code:
17      main:                           ;main function
18      save     %sp, -128, %sp         ;set the stack frame
19      mov      25, %g1
20      st       %g1, [%fp-20]          ;int a=25;
21      mov      7, %g1
22      st       %g1, [%fp-16]          ;int b=7;
23      ld       [%fp-20], %o0          ;put a in out register %o0
24      ld       [%fp-16], %o1          ;put b in out register %o1
25      call     foo, 0                 ;call function foo
26      nop
27      mov      %o0, %g1               ;move the returned value to %g1
28      st       %g1, [%fp-12]          ;store the value to [%fp-12]
29      restore                         ;restore caller's window
30      jmp      %o7+8                  ;jump back to the callers
31      nop
32
33      foo:
34      save     %sp, -112, %sp         ;set the stack frame
35      st       %i0, [%fp+68]          ;store a into [%fp+68]
36      st       %i1, [%fp+72]          ;store b into [%fp+72]
37      ld       [%fp+68], %g2          ;load a from memory space to %g2
38      ld       [%fp+72], %g1          ;load b from memory space to %g1
39      add      %g2, %g1, %g1          ;add a and b, store in %g1
40      st       %g1, [%fp-12]          ;store the result to [%fp-12]
41      ld       [%fp-12], %g1
42      mov      %g1, %i0               ;move the result from %g1 to %i0
43      restore                         ;restore caller's window
44      jmp      %o7+8                  ;jump back to the callers
45      nop
```

Then, on line 37 and 38, variables a and b are loaded from stack to %g1 and %g2. The LD instruction ensures the parameters are allowed to be used by the current thread, which has tag (user1, manager1). Line 39 adds the value of a and b, then stores the result (c1) in %g1. This is an arithmetic instruction, which needs to follow the rule for arithmetic instruction and the rules for store instructions (completing the addition operation on line 12). Line 40 stores the result to stack. Line 41 and 42 store the result in %i0, which is the place for the return value.

On line 43, RESTORE instruction restores the caller's window. Then, line 44 jumps to the address that is stored in %o7+8. For this jump, there is a check to make sure that the target address is tagged with the same tag of the previous PC (completing the return on line 13). Back to main function, the PC changes back to the previous value. Lines 27 and 28 store the returned value to [%fp-12] on the stack (completing the code on line 7).

Figure 4. Sample code

CONCLUSIONS

This paper proposed a new security tagging scheme that enhances access control through least privilege, while enabling good system performance. We chose RTEMS as our target system, as an exemplary ZKOS. Our tagging scheme has been designed to replace the classical supervisor/user mode operating model of operating systems and therefore protect the operating system code and data. The core pieces of our scheme focus on identifying controlling "code-space" and "owner's" for all code and data in the system, and providing appropriate access control. We validated our tagging scheme by extending the SPARC Instruction Simulator, to simulate the tag engine.

We still need to investigate performance enhancements to make it realizable in hardware. For example, the LOAD instruction normally loads values from memory space to registers, but in our tagging scheme, the LOAD instruction has to check the value's tag and store it as the register's tag additionally. To minimize overhead, we plan to cache as many tags as we can to speed up the tag checking operations and we are confident that we can use a tag compression scheme, and data and code spatial locality information to reduce the overhead.

Lessons Learned

Initial work on security tagging architecture seemed interesting; we could give the hardware the ability to help us enforce security by providing fine-grain protection. We used the 'C' programming language as the initial model of execution, giving us a high-level language approach to the security model, while being able to reason about the lower-level security operations. When we moved to implementation, we found we had to look at how the assembly language implementation actually worked. We found several issues that were missed at the higher level, and in the discussions of other tagging schemes in the literature:

- Compiler optimizations will change and/or remove security relevant code. For example, we could tag a small function with a high-level security tag, hoping to prevent user access to the tag. The compiler can then "in-line" the function, effectively moving the code into the user code space and ignoring the security tags. This was especially a problem when writing small test cases for evaluating the tagging and the simulator.
- Hardware features can change the execution and security model. The SPARC processor uses a register window to improve performance, and does not necessarily use the stack. Security models that assume access to the run-time stack may fail when the compiler does not implement a stack, but instead just uses the hardware features.
- Hardware researchers often use small test cases and then generalize their results. We found that testing with full commercial compilers and operating system can expose problems with some of the proposed solutions.
- History has shown us that programmers make mistakes. They will forget about adding security features, they will leave code vulnerable to attack or they will mislabel or misuse security tags. We have no clear indication that the addition of security tagging architecture will protect programmers from themselves, or that the added complexity of the tagging hardware control software will not make the security problem even worse.

ACKNOWLEDGEMENT

This work was funded in part by AFRL contract FA8750-11-2-0047. The opinions and findings expressed in this paper are solely those of the authors and do not represent the position of the U.S. Government or AFRL.

REFERENCES

1. Applications Research Corporation. RTEMS C User’s Guide, RTEMS 4.10.1 (online) edition, 2011.
2. D. Bell and L. LaPadula. Secure computer system unified exposition and Multics interpretation. *Technical Report MTR-2997*, MITRE Corp., Bedford, MA, 1975.
3. K. Biba. Integrity considerations for secure computer systems. *Technical Report MTR-3153*, *Rev. 1*, The Mitre Corporation, 1977.
4. M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. *In Intl. Symposium on Computer Architecture*, 482–493, 2007.
5. J. Fenton. Memoryless subsystems. *The Computer Journal*, *17(2)*, 1974.
6. H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. *IEEE/IFIP Intl. Conference on Dependable Systems and Networks*, 105–114, 2009.
7. F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. *Intl. Symposium on Microarchitecture*, 135–148, 2006.
8. J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63(19):1278–1308, 1975.
9. R. Shioya, D. Kim, K. Horio, M. Goshima, and S. Sakai. Low-overhead architecture for security tag. *IEEE Pacific Rim Intl. Symposium on Dependable Computing*, 135–142. IEEE Computer Society, 2009.
10. A. Shriraman and S. Dwarkadas. Sentry: Light-weight auxiliary memory access control. *Intl. Symposium on Computer Architecture*, 407–418, 2010.
11. H. Shrobe, A. DeHon, and T. Knight. Trustmanagement, intrusion tolerance, accountability, and reconstitution architecture (tiara). *AFRL Technical Report AFRL-RI-RS-TR-2009-271*, 2009.

12. J. Song. Development and evaluation of a security tagging scheme for a real-time zero operating system kernel. *Master's thesis*, University of Idaho, 2012.
13. J. Song and J. Alves-Foss. Security Tagging for a Zero-Kernel Operating System, in *HICSS* January, 2013.
14. G.E. Suh, J.W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, 85–96, 2004.
15. E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. *Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, 304–316, 2002.
16. S. Hsi Yong and S. Horwitz. Protecting C programs from attacks via invalid pointer dereferences. *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 307–316, 2003.
17. N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. *USENIX Symposium on Operating Systems Design and Implementation*, 225–240, 2008.